Estratégias de Desenvolvimento de Algoritmos- Subsequência Comum Mais Longa (“*The Longest Common Subsequence Problem*”)

Ana Sofia Medeiros de Castro Moniz Fernandes

Resumo **- O presente relatório tem como propósito a apresentação e explicação do problema *Subsequência Comum Mais Longa* (“*The Longest Common Subsequence Problem*”), no âmbito do primeiro trabalho prático da Unidade Curricular Algoritmos Avançados, do Mestrado de Engenharia Informática. Ao longo do documento, serão apresentados e explicados os detalhes do problema (resolvido com três diferentes algoritmos), bem como alguns testes realizados e as suas interpretações.**

Abstract **- The present report aims to present and explain the problem *The Longest Common Subsequence Problem,* in the context of the first practical work of the course Algoritmos Avançados of the Informatics Engineer Master’s Degree. Over the document, it will be presented and explained all the constituent details of the problem (solved with three different algorithms), as well as some performed tests and its interpretations.**

I. INTRODUÇÃO

No primeiro trabalho prático da Unidade Curricular de Algoritmos Avançados, foi proposta aos alunos a escolha de um tema de entre dezanove, que se encontram divididos em dois grupos - pesquisa exaustiva e programação dinâmica.

O tema escolhido e apresentado, Subsequência Comum Mais Longa**,** pertence ao grupo de programação dinâmica. Assim sendo, foram desenvolvidos e testados três algoritmos - um algoritmo recursivo, um algoritmo recursivo usando "memoization" e um algoritmo dinâmico. Além disso, foi realizada uma análise da eficiência computacional e das limitações dos algoritmos desenvolvidos.

II. PROBLEMA DA SUBSEQUÊNCIA COMUM MAIS LONGA

Dadas duas sequências, o problema da subsequência comum mais longa passa, nada mais nada menos, por encontrar a subsequência comum mais longa entre as sequências. É de notar que uma subsequência é uma sequência de carateres (incluindo espaços e símbolos) que aparecem na mesma ordem relativa, mas não necessariamente de forma contínua. Como se pode observar no exemplo dado pela figura Fig.1 [7], dadas duas sequências "nematode knowledge" e "empty bottle", pode encontrar-se a subsequência mais longa desenhando-se traços (não cruzados) entre as duas sequências. Uma palavra de comprimento *n* tem, assim, 2^n subsequências.

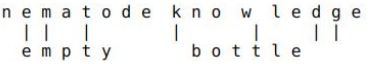

Fig. 1 - Subsequência mais comum entre as sequências “nematode knowledge” e “empty bottle”.

III. ANÁLISE E INTERPRETAÇÃO DA IMPLEMENTAÇÃO DOS DIFERENTES ALGORITMOS

A. Algoritmo recursivo

Por forma a resolver o problema usando-se programação dinâmica, deve começar-se por resolvê-lo através do algoritmo recursivo - a programação dinâmica não nos ensina a chegar à solução, pois apenas irá tornar a solução inicial mais eficiente [7].

Observando novamente a figura Fig.1 [7], é possível entender que:

- Se duas sequências começarem com o mesmo caracter, é seguro escolher este caracter para ser o primeiro da subsequência [7] - vai ser possível traçar a linha imaginária, entre as sequências, sem que haja sobreposição com outras linhas;
- Se duas sequências não começarem com o mesmo caracter, não é possível que ambos os caracteres façam parte da sequência [7] - pelo menos um terá que ser eliminado.

Uma vez que o objetivo principal é encontrar o comprimento da subsequência mais longa, e não a subsequência em si, pode determinar-se qual o subproblema que dá a melhor solução através de recursividade. Por forma a evitar-se um grande número de ciclos, poderá começar-se a analisar a sequência pelo fim, comparando os caracteres de cada palavra na posição “comprimento-1” de cada uma. [7]

Para uma melhor percepção, passe a analisar-se o fluxograma da abordagem recursiva:

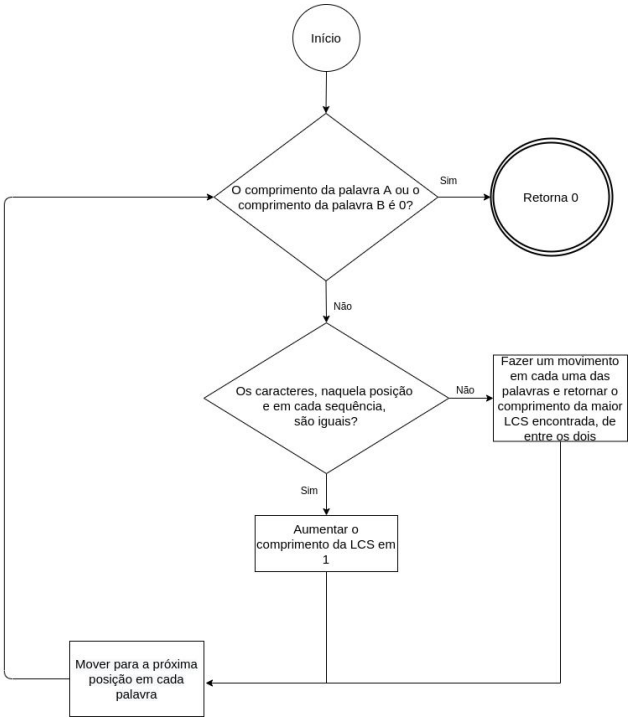

Fig. 2 - Fluxograma para o algoritmo recursivo

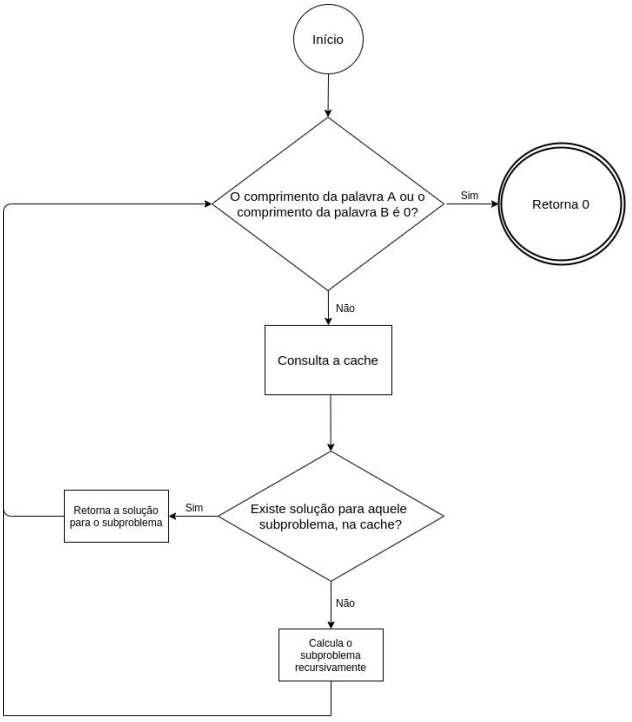

Fig. 3 - Fluxograma para o algoritmo recursivo, usando "memoization"

• Se o comprimento de alguma das sequências for 0, é retornado 0 e o programa termina [3];
• Se os caracteres, naquela posição e em cada sequência forem iguais, aumenta-se o valor do comprimento da subsequência comum mais longa em uma unidade, e prossegue-se recursivamente para a próxima posição em cada sequência [3];
• Se os caracteres, naquela posição e em cada sequência forem diferentes, faz-se um movimento na sequência A e um na sequência B, e seguidamente retorna-se o máximo devolvido entre a chamada recursiva para estes dois movimentos [3].

Assim, em pseudo-código [5]:

```
LCSLength(X[1..m], Y[1..n], m, n)
    if m=0
        return 0
    if n=0
        return 0
    if X[i-1] = Y[j-1]
        return 1 + LCSLength(X,Y,m-1,n-1)
    else
        max(LCSLength(X,Y,m,n-1),
           LCSLength(X,Y,m-1,n))
```

B. Algoritmo recursivo usando "memoization"

Para implementação do algoritmo recursivo, recorrendo a "memoization", foi usado um “wrapper” [2]. Este “wrapper” tem como finalidade evitar que a função recursiva seja chamada para subproblemas que já haviam sido resolvidos. Para isso, armazena-os numa cache e, cada vez que os subproblemas voltarem a surgir, basta aceder à cache, evitando-se assim chamadas recursivas para resolver mais uma vez aquele problema.

C. Algoritmo de programação dinâmica

A implementação com programação dinâmica permite melhorar a complexidade do problema, uma vez que evita o cálculo de subproblemas repetidos e chamadas recursivas.

Isto é possível devido ao uso de um "array" de tamanho m+1 por n+1 (sendo m o tamanho da primeira sequência e n o tamanho da segunda).

Este "array", inicializado com todas as posições a -1, vai permitir guardar sucessivamente o tamanho das subsequências encontradas:

• Na primeira iteração do ciclo, começa-se por preencher a primeira linha e primeira coluna do "array" com zeros, por forma a poder-se começar a realizar cálculos acerca do tamanho das subsequências [3][8];
• Se os caracteres naquela posição e em cada sequência forem iguais, coloca-se, naquela posição do "array", a soma entre o tamanho da subsequência da célula anterior do "array" e uma unidade [3];
• Se os caracteres, naquela posição e em cada sequência forem diferentes, faz-se um movimento na sequência A e um na sequência B, e seguidamente coloca-se o valor máximo do comprimento encontrado, entre estas duas posições, no "array" [3].

Para uma melhor percepção, passe a analisar-se o fluxograma da abordagem com programação dinâmica:

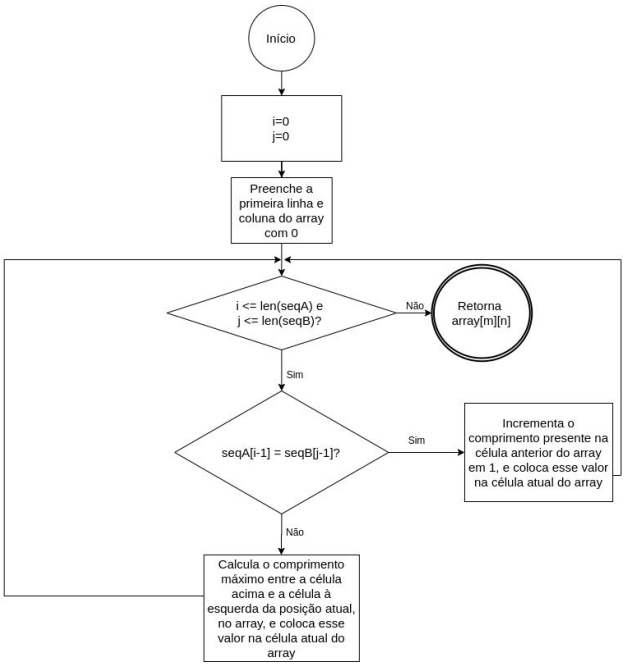

Fig. 4 - Fluxograma para o algoritmo de programação dinâmica

Assim, em pseudo-código:

```
function LCSLength(X[1..m], Y[1..n])
    table = "array"(0..m, 0..n)
    for i := 0..m
        table[i,0] = -1
    for j := 0..n
        table[0,j] = -1
    for i := 0..m
        for j := 0..n
           if i==0 || j==0
                table[i,j] = 0
            if X[i-1] = Y[j-1]
                table[i,j] := 1 +
                           table[i-1,j-1]
            else
                table[i,j] :=
                  max(table[i,j-1],
                  table[i-1,j])
    return table[m,n]
```

Na implementação com programação dinâmica, devido ao facto de ser usado um "array" onde são guardados os sucessivos tamanhos das subsequências encontradas, é possível criar-se uma função recursiva para devolver a(s) subsequência(s) mais longa encontrada(s), através do "traceback" do "array". Quando o tamanho diminui entre duas células do "array", poderá ser um sinal de que aquelas subsequências têm esse elemento em comum [5][9]. Começa-se por criar um set vazio e, após isso, verifica-se [5][9]:

• Se o comprimento de alguma das sequências for 0, adiciona-se “” ao set;

• Se os caracteres, naquela posição e em cada sequência forem iguais, significa que esse caracter deve estar presente em todas as subsequências possíveis. Ao set é adicionado a concatenação entre o caracter em questão e todas as subsequências possíveis (encontradas através da chamada recursiva à função) ;

• Se os caracteres, naquela posição e em cada sequência forem diferentes, a subsequência comum mais longa deve ser construída através da direção acima ou à esquerda, no "array" - escolhe-se aquele que tiver um maior tamanho armazenado, ou ambos se tiverem o mesmo valor, e adiciona-se ao set.

Por fim, é retornado o set, que contém todas as subsequências mais longas possíveis, cada uma escrita de forma inversa, dado que foi feito um "traceback" ao "array".

IV. ANÁLISE DA COMPLEXIDADE DO PROBLEMA

A. Algoritmo recursivo

Para encontrar a complexidade da aproximação recursiva, é necessário ter conhecimento acerca do número de possíveis subsequências de uma string de tamanho n - é necessário encontrar o número de subsequências com tamanhos de 1 a n-1. Para um elemento, o número de combinações serão nC1. Como nC0 + nC1 + nC2 +... + nCn = 2^n , a aproximação recursiva terá uma complexidade de O(2^n), que poderá ser melhorada com programação dinâmica [4].

No pior dos casos, na programação recursiva, a complexidade será também de O(2^n), e o pior caso acontece quando todos os caracteres de uma sequência são diferentes dos da outra - o tamanho da subsequência comum mais longa será zero.

B. Algoritmo recursivo usando memoization

A complexidade deste algoritmo não é tão boa como a de programação dinâmica, uma vez que ainda realiza chamadas recursivas, mas também não é tão má quanto a do algoritmo recursivo, uma vez que usa uma cache para aceder à solução de subproblemas que já haviam aparecido.

C. Algoritmo de programação dinâmica

Com programação dinâmica, o tempo de execução será len(sequênciaA)*len(sequênciaB). Assumindo que m=len(sequênciaA) e n=len(sequênciaB), a complexidade deste algoritmo será de O(m*n). [3]

V. ANÁLISE DOS RESULTADOS

Por forma a comparar os desempenhos dos três algoritmos, foram criados 8 exemplos, onde as instâncias vão aumentando sucessivamente de dimensão, de exemplo para exemplo:

•Example1.py: a subsequência mais longa tem comprimento 3;

•Example2.py: a subsequência mais longa tem comprimento 8;

•Example3.py: a subsequência mais longa tem comprimento 25;
•Example4.py: a subsequência mais longa tem comprimento 68;
•Example5.py: a subsequência mais longa tem comprimento 486;
•Example6.py: a subsequência mais longa tem comprimento 1287;
•Example7.py: a subsequência mais longa tem comprimento 3345;
•Example8.py: a subsequência mais longa tem comprimento 6393.

A. Desempenho do algoritmo recursivo

No algoritmo recursivo apenas foi possível a execução dos primeiros dois exemplos. O exemplo 3 foi deixado a correr durante mais de oito horas, não sendo este tempo o suficiente. Estime-se, na melhor das hipóteses, um tempo de 10 horas para o exemplo 3. Assim:

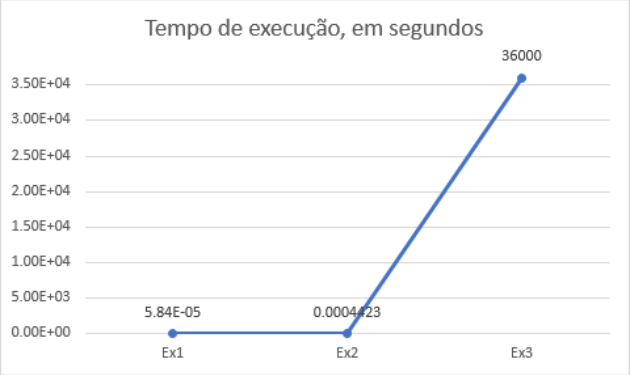

Fig. 5 - Tempo de execução do algoritmo recursivo, para os primeiros três exemplos

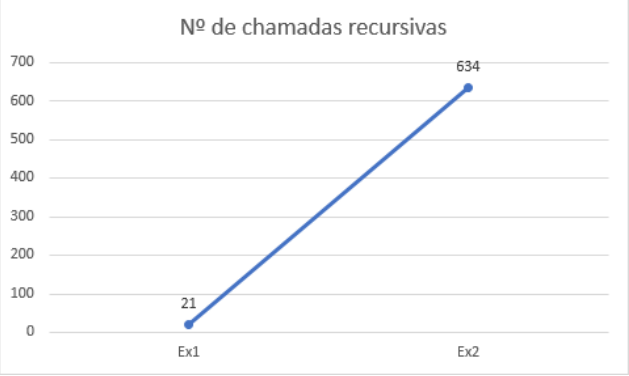

Fig. 6 - Número de chamadas recursivas feitas pelo algoritmo recursivo, para os primeiros dois exemplos

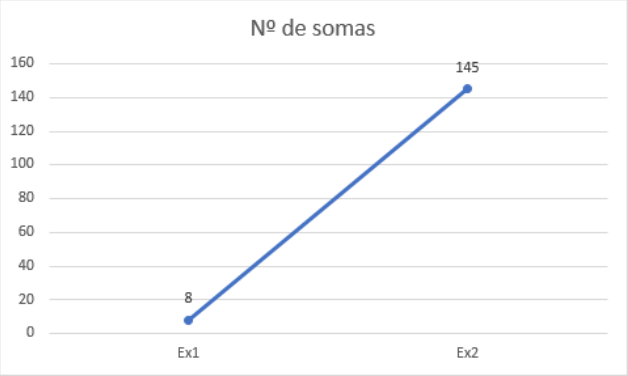

Fig. 7 - Número de somas calculadas pelo algoritmo recursivo, para os primeiros dois exemplos

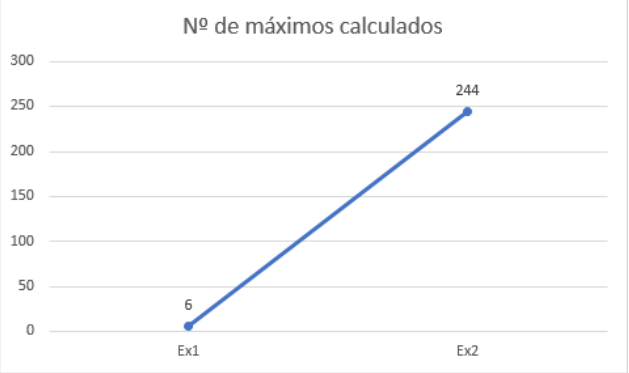

Fig. 8 - Número de máximos calculados pelo algoritmo recursivo, para os primeiros dois exemplos

É possível concluir-se que, no algoritmo recursivo, quando uma instância cresce, o tempo de execução, o número de chamadas recursivas e o número de operações básicas também cresce, mas de forma exponencial. Isto deve-se ao facto de, no algoritmo recursivo, serem calculados de forma repetida os mesmos subproblemas. É o algoritmo que apresenta um maior número de operações básicas e tempo de execução mais longo.

B. Algoritmo recursivo usando "memoization"- número de acessos à cache

Como dito anteriormente, no algoritmo recursivo usando "memoization" usa-se uma cache, que guarda a solução dos subproblemas já resolvidos, por forma a não ter que calcular recursivamente resultados de subproblemas que já haviam sido resolvidos.

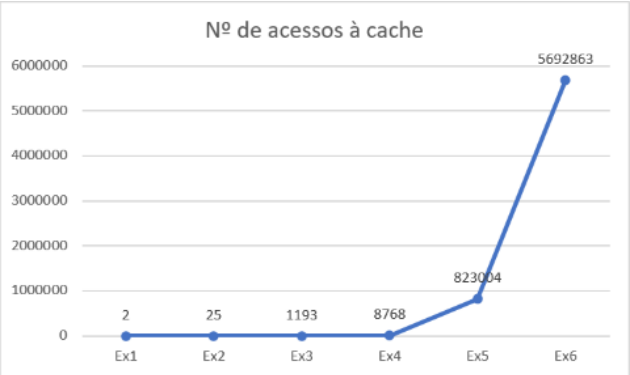

Fig. 9 - Número de acessos à cache, no algoritmo recursivo usando "memoization"

Pela análise do gráfico, é possível ver que o número de acessos à cache cresce à medida que a complexidade do problema aumenta - vão existindo cada vez mais subproblemas cuja solução já havia sido calculada.

C. Algoritmo recursivo usando "memoization" vs algoritmo de programação dinâmica

Por forma a compreender-se melhor o desempenho destes dois algoritmos, foram traçados gráficos para o número de operações básicas e para o tempo de execução de cada um, em segundos.

É de notar que, relativamente aos exemplos 7 e 8, não são possíveis de executar com o algoritmo de "memoization". Mesmo definindo o limite de recursão, em "Python3", para quinze mil, é apresentado um erro de "segmentation fault".

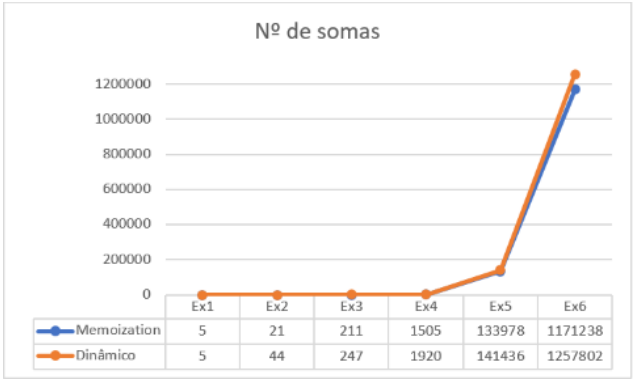

Fig. 10 - Comparação do número de somas calculadas pelos dois algoritmos, para os primeiros seis exemplos

Fig. 11 - Comparação do número de máximos calculados pelos dois algoritmos, para os primeiros seis exemplos

Quanto ao número de operações básicas de cada algoritmo, é possível verificar que o algoritmo de "memoization", apesar de mais lento, realiza um número menor de operações básicas (apesar de não ser uma diferença muito notória). Isto poderá ser devido ao facto de usar uma cache - apesar de realizar menos operações, acaba por ser mais lento pois, se não tiver a solução em cache, terá que a calcular de forma recursiva, o que acaba por atrasar todo o processo.

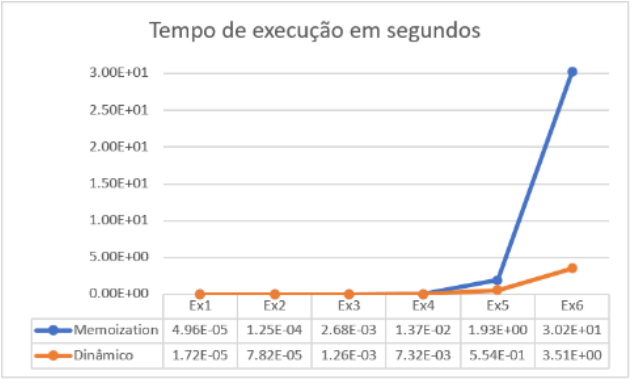

Fig. 12 -Comparação de tempos de execução dos dois algoritmos, para os primeiros seis exemplos

Através da análise do gráfico acima, é de fácil percepção que, conforme as instâncias dos exemplos aumentam de dimensão, o tempo de execução para o algoritmo de "memoization" se torna bastante maior que o tempo de execução necessário para o algoritmo de programação dinâmica. Nos primeiros três exemplos, onde a dimensão ainda é relativamente pequena, a diferença não é tão acentuada, começando esta a aumentar mais notoriamente a partir do exemplo 4. No exemplo 6, a diferença acaba mesmo por ser de cerca de 27 segundos.

Apesar do algoritmo recursivo usando "memoization" não ser capaz de executar os exemplos 7 e 8, o algoritmo dinâmico fá-lo com bastante facilidade, demorando apenas cerca de 26 e 96 segundos, respetivamente. Isto deve-se ao facto de usar um "array", tendo sempre a solução dos subproblemas armazenadas. Mesmo no caso de não ter, não a calcula de forma recursiva - apenas necessita de aceder às células anteriores àquela posição, no "array".

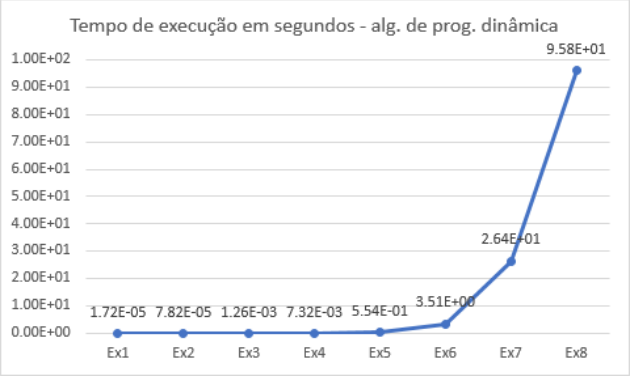

Fig. 13 - Tempo de execução, em segundos, para o algoritmo de programação dinâmica

D. Algoritmo de programação dinâmica - encontrar a subsequência comum mais longa

Como mencionado anteriormente, no algoritmo de programação dinâmica é possível encontrar a *string* representativa da subsequência comum mais longa, através duma função recursiva, onde se realiza o "traceback" do "array". As subsequências comuns mais longas encontradas foram:

•Exemplo 1: *ACD* e *ABD*

•Exemplo 2: *AANAAANA*

•Exemplo 3: *GTABGTABGTABGTABGTABGTABG*

O exemplo 4 foi deixado a correr durante mais de 5 horas, não tendo o programa acabado de executar nesse tempo. Estima-se, assim, um tempo de 6 horas, na melhor das hipóteses. Ora:

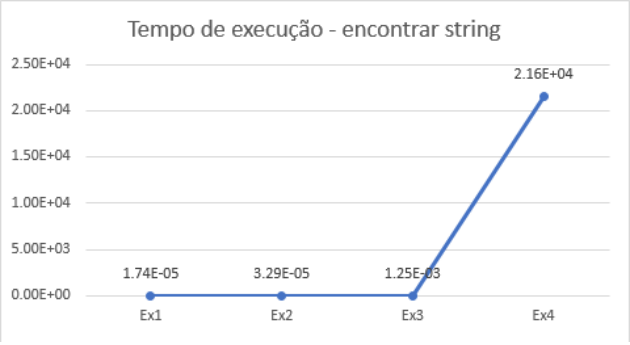

Fig. 14 - Tempo de execução, em segundos, para o encontrar das subsequências mais longas, no algoritmo de programação dinâmica

Conclui-se desta forma que, como seria de esperar, quanto maior forem as subsequências encontradas, maior será o tempo de execução.

VI. CONCLUSÃO

Tendo em conta o estudado neste relatório, é possível inferir que:

• A implementação do algoritmo de programação dinâmica deve sempre ser realizada após a implementação da resolução recursiva, uma vez que a dinâmica não dará a solução do problema, apenas a vai melhorar;

• A complexidade de um problema, quando usada programação dinâmica, diminui bastante, tornando o programa mais eficiente;

• O número de operações básicas realizadas no algoritmo recursivo cresce muito mais rapidamente do que nos restantes algoritmos;

• Apesar do algoritmo de programação dinâmica apresentar tempos de execução bastante melhores que o algoritmo de "memoization", este último apresenta um número menor de operações básicas;

• Apenas é possível encontrar a subsequência comum mais longa através do algoritmo de programação dinâmica, uma vez que é possível fazer "traceback" do "array".

REFERÊNCIAS

[1] J. Madeira, "Estratégias Algorítmicas", Aveiro, 2020
[2] J. Madeira, "Programação Dinâmica", Aveiro, 2020
[3] "Longest Common Subsequence (LCS) - Recursion and Dynamic Programming", [Online]. Available: https://tinyurl.com/yyv739lg [Acedido em 6 Dezembro 2020].
[4] "Longest Common Subsequence | DP-4", [Online]. Available: https://tinyurl.com/y2h8acmh [Acedido em 6 Dezembro 2020].
[5] "Longest common subsequence problem", [Online]. Available: https://tinyurl.com/86bxp9f [Acedido em 6 Dezembro 2020].
[6] "Intro to memoization and dynamic programming (LCS)", [Online]. Available: https://tinyurl.com/yx9b5olz [Acedido em 6 Dezembro 2020].
[7] "ICS 161: Design and Analysis of Algorithms Lecture notes for February 29, 1996", [Online]. Available: https://tinyurl.com/nayubac [Acedido em 6 Dezembro 2020].
[8] "Python Program for Longest Common Subsequence", [Online]. Available: https://tinyurl.com/y5sdc4ec [Acedido em 6 Dezembro 2020].
[9] "Printing Longest Common Subsequence | Set 2 (Printing All)", [Online]. Available: https://tinyurl.com/y24t5xb5 [Acedido em 6 Dezembro 2020].
[10] "1: LCS Algorithm Flowchart", [Online]. Available: https://tinyurl.com/yypbmryy [Acedido em 6 Dezembro 2020].